ISIMA
Institut
Supérieur d'
Informatique de
Modélisation et de leurs
Applications

GERAD
Groupe d'
Etude et de
Recherche en
Analyse des
Décisions

Rapport de stage de deuxième année d'école d'ingénieur
Filière 2 : génie logiciel et système informatique

Classification de données non supervisée

Tome I

Auteur :
Etienne Duclos

Durée : **5 mois**
Avril à Septembre 2009

Responsables Entreprises :
Gilles Caporossi
Pierre Hansen
Sylvain Perron
Responsable ISIMA :
Christophe Duhamel

2008 – 2009

Institut
Supérieur d'
Informatique de
Modélisation et de leurs
Applications

Groupe d'
Etude et de
Recherche en
Analyse des
Décisions

Rapport de stage de deuxième année d'école d'ingénieur
Filière 2 : génie logiciel et système informatique

Classification de données non supervisée

Tome I

Auteur :
Etienne Duclos

Durée : **5 mois**
Avril à Septembre 2009

Responsables Entreprises :
Gilles Caporossi
Pierre Hansen
Sylvain Perron
Responsable ISIMA :
Christophe Duhamel

2008 – 2009

Remerciements

Je tiens avant tout à remercier toutes les personnes qui m'ont permis d'effectuer mon stage dans un environnement de travail agréable et dans d'excellentes conditions. Je tiens ainsi à remercier tous les employés du GERAD pour leur accueil au sein de leur laboratoire.

Je remercie aussi chaleureusement Messieurs Gilles Caporossi, Pierre Hansen et Sylvain Perron, mes maîtres de stage, pour leur accueil, leur disponibilité, ainsi que leurs conseils et leur encadrement tout au long du stage. Je tiens également à remercier Monsieur Christophe Duhamel, mon référent ISIMA, pour m'avoir donner l'opportunité de faire ce stage.

Je tiens enfin à remercier Mme Mouzat pour ses cours de communication qui m'ont aidé à faire ce rapport et à préparer ma soutenance.

Table des figures

Figure 1 : Problème de départ....5
Figure 2 : problème relaxé....6
Figure 3 : Résolution du problème linéaire....7
Figure 4 : Problème auxiliaire....7
Figure 5 : Diagramme de classe UML de l'application....9
Figure 6 : Ligne de commande du main global....12
Figure 7 : Exemple d'un fichier de profilage généré par Gprof....13
Figure 8 : Ancien diagramme de classe UML de la partie solver....14
Figure 9 : Nouveau diagramme de classe UML de la partie solver....14
Figure 10 : En-tête de la classe Example....16
Figure 11 : Diamètre d'un cluster....19
Figure 12 : Problème de la somme des diamètres....20
Figure 13 : Algorithme initial de la méthode calculDiam....20
Figure 14 : Nouvel algorithme de la méthode calculDiam....21
Figure 15 : Algorithme initial de la méthode CreateFirstColumn....22
Figure 16 : Algorithme temporaire de la méthode CreateFirstColumn....22
Figure 17 : Algorithme final de la méthode CreateFirstColumn....23
Figure 18 : Comparaison de résultats....24
Figure 19 : Algorithme de la descente....25
Figure 20 : Algorithme de la V.N.S....26
Figure 21 : Algorithme de la méthode init 27
Figure 22 : Problème linéaire associé à la méthode exacte....28
Figure 23 : Ancien code de la méthode exacte....28
Figure 24 : Nouveau code de la méthode exacte....29
Figure 25 : Algorithme de la méthode hiérarchique....38
Figure 26 : Algorithme de construction de la liste des NN....38
Figure 27 : Exemple de dendogramme....39
Figure 28 : Algorithme de la méthode run....40
Figure 29 : Algorithme de la méthode DetermineNN....41
Figure 30 : Résultat de la méthode hiérarchique....41

Résumé

La **classification de données non supervisée** est une méthode consistant à classer différents objets en une partition de sous ensembles, non connue à l'avance, selon un critère. Ces observations peuvent ainsi être classées de manière homogène, distincte, ou les deux. L'homogénéité se traduit par le fait que tous les points présents dans un sous ensemble sont similaires ; la distinction se traduit quand à elle par le fait que les points présents dans un sous ensemble sont différents de ceux présents dans un autre sous ensemble.

Le critère de **la somme des diamètres** est un problème de classification homogène. Selon ce critère les objets sont rangés dans les sous ensembles de telle sorte que le diamètre de la partition finale, qui correspond à la **somme des diamètres** des sous ensembles la composant, soit le plus petit possible. Le diamètre d'un sous ensemble correspond à la plus grande dissimilarité entre deux objets présents dans cet ensemble.

Pour résoudre de tels problèmes, le programme sur lequel j'ai effectué mon stage, et qui se compose d'une **heuristique** initiale, d'une seconde **heuristique** et d'une **méthode exacte**, utilise des **logiciels de programmation linéaire**, reliés au programme grâce à des classes spécifiques. L'utilisation de tels logiciels peut cependant poser des problèmes, au niveau par exemple de la **précision numérique**, et nous devons vérifier que celle-ci est la même dans tout le programme, en utilisant une classe dédiée à cette tâche.

Il existe aussi une manière rapide de connaître le nombre de sous ensembles optimal pour un problème donné. Il suffit pour cela d'appliquer une **méthode hiérarchique** à ce problème. Une telle méthode nous fournie le coût engendré par la fusion de deux sous ensembles, et nous permet de déterminer si une classification en ce nombre d'ensemble est correcte ou non.

Mots clés : classification de données non supervisée, somme des diamètres, heuristique, méthode exacte, logiciel de programmation linéaire, précision numérique, méthode hiérarchique

Abstract

The **clustering** is a method which consist in classifying different observations in an unknown partition of clusters following a criterion. These observations can be classified in an homogeneous way, a distinct way, or those two. The homogeneity is when every points in a same cluster are very close, and the distinction is when the points in a cluster are different from the points in other clusters.

The **sum of diameters** criterion is an homogeneous problem of clustering. Following this criterion, points are classified in clusters in order to have the smallest diameter for the final partition, which means that the sum of all the diameters of the clusters in the partition are is the smallest possible.

In order to solve such problems, the program on which i did my internship, and which is composed by an initial **heuristic**, a second **heuristic** and an **exact method**, use **linear programming softwares**. However these softwares can add problems, such as the **numerical precision**, and we have to use a specific class in order to control this.

There is also a fast way to know the optimal number of cluster for a given problem. We have to use a **hierarchic method**, which gives us the cost of the fusion of two clusters and allow us to determine if a classification in that number of cluster is correct or not.

Keywords : clustering, sum of diameters, heuristic, exact method, linear programming software, numerical precision, hierarchic method

Remerciements

Table des figures

Résumé

Abstract

Table des Matières

Introduction 1

1 Présentation 3
1.1 Présentation du GERAD....3
1.2 Présentation du problème....4
1.2.1 La classification non supervisée....4
1.2.2 Utilisation du Branch And Bound....5
1.2.3 Méthode de résolution par génération de colonne....6
1.2.4 Le projet existant....7

2 Gestion du code 11
2.1 Passage de 32 à 64 bits....11
2.2 Utilisation de la classe accuracy....11
2.3 Main Global....12
2.4 Gprof....12
2.4.1 Description....12
2.4.2 Utilisation....13
2.5 Création de classes....14
2.5.1 Création de classes mères pour les solver....14
2.5.2 Création de la classe Example....15
2.5.3 Utilisation d'une partition....17

3 La somme des diamètres 19
3.1 Modifications du code existant....20
3.1.1 Optimisation de méthodes....20
3.1.2 Modification de méthodes....21
3.1.3 Changement de type de contraintes du problème maitre....23
3.2 Nouvelle heuristique de départ : la V.N.S....25
3.3 Méthode exacte de résolution du problème auxiliaire....27
3.3.1 Passage à GLPK 28
3.3.2 Résultats....29

4 Glpk 32
4.1 Description....32
4.2 Implémentation du solver....33
4.2.1 Sans stabilisation....33
4.2.2 Avec stabilisation....34
4.3 Résultats du solver....34

5 Méthode hiérarchique 37
5.1 Principe et code existant....37

5.1.1 Principe....37
5.1.2 Code existant....38
5.2 Implémentation....39
5.3 Résultats....41

Conclusion **43**

Références bibliographiques

Lexique de programmation

Lexique du programme

Introduction

Le monde de l'industrie, de part sa volonté de cibler ses clients potentiels, est très intéressé par les problèmes de classification de données. Ainsi, lors d'une étude de marché, une entreprise cherchera à connaître les groupes de population les plus susceptibles d'acheter son produit. Pour cela elle va établir différents critères, tel l'âge ou le sexe de la personne, qui lui permettront par la suite de classer cette personne dans un groupe au moyen de méthodes de classification.

Mon stage de deuxième année à l'Institut Supérieur d'Informatique, de Modélisation et de leurs Applications (ISIMA), effectué au Groupe d'Etude et de Recherche en Analyse des Décisions (GERAD) à Montréal, portait sur un programme de classification de donnée non supervisée. Le but du stage était, en collaboration avec les quatre autres stagiaires, de continuer l'implémentation de ce programme. Mon travail a été dans un premier temps de continuer la partie concernant le critère de la somme des diamètres, de m'occuper de la précision numérique dans le code et de créer un programme permettant d'effectuer une méthode hiérarchique indépendamment du projet. Cependant, au fil des semaines, j'ai été amené en plus à implémenter un lien entre le programme et un logiciel de programmation linéaire et à m'occuper de la maintenance du code.

Je présenterai dans une première partie l'environnement de travail ainsi que le projet existant. Dans une deuxième partie seront expliqués les changements que j'ai réalisé dans le code du programme. La troisième partie décrira le critère de la somme des diamètres et les modifications que j'ai apporté à ce sous problème. J'expliquerai dans une quatrième partie l'ajout de la possibilité d'utiliser le logiciel de programmation linéaire Glpk. Enfin la dernière partie traitera de la création d'un programme effectuant une méthode hiérarchique.

Première partie

Présentation

1 Présentation

1.1 Présentation du GERAD

Le GERAD, Groupe d'Etude et de Recherche en Analyse des Décisions, est un centre de recherche inter-universitaire de Montréal créé en 1979. Il est essentiellement financé par HEC Montréal, l'école Polytechnique de Montréal, l'Université McGill, l'Université du Québec à Montréal et le Fond Québécois de la Recherche sur la Nature des Technologies (FQRNT).

Il regroupe actuellement 70 chercheurs spécialistes en méthodes quantitatives de gestion, en recherche opérationnelle, en informatique théorique et en mathématiques. Il compte aussi 37 chercheurs post-doctoraux. Le GERAD publie aussi chaque année environ 70 numéros des « Cahiers du GERAD », regroupant les derniers travaux de ses membres. De plus ses chercheurs obtiennent régulièrement des contrats de recherche provenant d'entreprises privées, du Québec et d'ailleurs, ainsi que d'organismes publics provinciaux ou fédéraux. Pour répondre à ces contrats le GERAD dispose d'équipes de chercheurs et de professeurs qui travaillent avec leurs élèves et en collaboration avec d'autres laboratoires, du Canada ou étrangers, et plus spécialement en Europe et aux États-Unis.

On compte parmi ses membres des chercheurs de renommée internationale, tels Vašek Chvátal, Pierre Hansen, Richard Loulou et François Soumis.

Vašek Chvátal est titulaire de la Chaire de recherche du Canada en optimisation combinatoire, et fut le lauréat en l'an 2000 du prix « Beale-Orchard-Hays for Excellence in Computational Mathematical and Programming ».

Pierre Hansen a été directeur du GERAD de 1996 à 2001, et est titulaire de la Chaire HEC en exploitation des données. Il est spécialiste en recherche opérationnelle et en théorie des graphes et a publié environ 300 articles. Il est aussi membre du comité de rédaction de nombreuses revues scientifiques internationales.

Richard Loulou a été directeur du GERAD de 1989 à 1992 et a fait partie de l'équipe qui a élaboré un logiciel de simulation utilisé par Al Gore dans ses travaux lui ayant valu son prix Nobel de la Paix en 2007.

François Soumis a été directeur du GERAD de 1992 à 1996 et fait partie de l'équipe à l'origine du logiciel GENCOL (logiciel de GENération de COLonnes). Il s'agit d'un logiciel « d'optimisation mathématique développé pour traiter les problèmes de tournées et d'horaires de véhicules et d'équipage » (Site internet de GENCOL). Il est utilisé pour mettre au point les systèmes de transport en commun dans des villes telles Tokyo, Singapour ou Barcelone. Ce logiciel est également utilisé par des compagnies telles Air France ou Air Canada.

Le GERAD travaille principalement sur trois grands axes de recherches qui sont les suivants :

- « Méthodes d'analyse mathématique pour l'aide à la décision » ;
- « Développement d'applications dans les grands systèmes technologiques, commerciaux et économiques » ;
- « Développement de logiciels commerciaux d'aide à la décision » ([1]).

Le laboratoire possède de plus trois « missions » :

- « Développer la mathématique de la décision sous toutes ses formes dans les grands systèmes technologiques, commerciaux, et économiques, et en amont de la décision, développer la modélisation fondée sur la statistique, la simulation et l'exploitation des données » ;
- « Former des chercheurs dans les domaines ci-dessus » ;
- « Contribuer à l'enrichissement de la société par la création de spins offs, et par le transfert de savoir faire sous forme de consultations en entreprise et de développement de logiciels d'aide à la décision » ([1]).

Cette volonté d'ouverture sur le monde de la recherche se traduit entre autre par l'organisation annuelle d'évènements tels « Les journées de l'optimisation », durant lesquelles des spécialistes mondiaux viennent présenter leurs recherches, ou « l'Université d'été du GERAD », série d'exposés de chercheurs venus d'universités américaines ou canadiennes. De plus les chercheurs du GERAD prennent part à « l'atelier de résolution de problèmes industriels », organisé par le Centre de Recherche en Mathématiques (CRM). Lors de ces ateliers, des personnes du monde de l'industrie soumettent des problèmes aux participants, qui sont répartis en équipes de travail comportant des chercheurs, des étudiants, et des personnes de l'entreprise concernée. Au bout d'une semaine, les équipes présentent leurs solutions à l'assemblée. Ces ateliers permettent de créer des liens entre le monde industriel et les chercheurs, et aboutissent à des contrats de recherche ou des sujets de thèses.

L'équipe de développement à laquelle j'étais intégré est composée d'élèves ingénieurs de l'ISIMA encadrés par trois chercheurs du GERAD spécialisés en recherche opérationnelle et optimisation. Le premier d'entre eux est Gilles Caporossi, co-auteur du logiciel de théorie des graphes AGX, co-responsable des séminaires du GERAD et professeur à HEC Montréal. Le second est Pierre Hansen, expert entre autres dans les problèmes de localisation tel la P-médiane. Le troisième est Sylvain Perron, professeur à HEC Montréal et développeur d'une nouvelle version du logiciel d'optimisation quadratique non convexe QP.

1.2 Présentation du problème

1.2.1 La classification non supervisée

Une part importante du travail d'une entreprise est l'étude de marché. En effet, avant de concevoir un produit en grande quantité il est toujours bon de savoir si celui ci se vendra bien ou non, à quelle population, et quelle publicité

serait la plus adéquate pour ce produit. Pour répondre à ces questions, les entreprises peuvent être amenées à classer leurs clients potentiels, selon par exemple leurs habitudes de consommation. Il s'agit alors de classification. On parle de classification non supervisée si les classes finales de répartition des « objets » ne sont pas connues par avance, et de classification supervisée dans le cas contraire. Le projet sur lequel portait mon stage traitait de la classification non supervisée.

Le but de la classification est de regrouper un ensemble d'objets, ou d'observations, en une partition de différents sous ensembles, qui doivent être homogènes et/ou distincts, selon un critère choisi. Les sous ensembles sont appelés classe en français et cluster en anglais. Cependant je vais par la suite utiliser le terme anglais cluster pour ne pas créer de confusion avec le terme classe qui est aussi utilisé lorsque l'on parle de programmation.

Des clusters sont homogènes si les observations à l'intérieur d'un même cluster se ressemblent. Des clusters sont distincts si les objets présents dans un cluster sont différents de ceux présents dans un autre cluster.

En pratique cela revient à minimiser une fonction objectif, qui est la somme du coût des clusters, sous certaines contraintes, comme nous pouvons le voir sur la figure suivante :

$$min \sum_{t \in T} f(C_t) y_t \quad (P)$$

$$sous\, contraintes: \quad \sum_{t \in T} a_{it} y_t = 1 \quad (1)$$

$$\sum_{t \in T} y_t = p \quad (2)$$

$$y_t \in \{0,1\} \quad (3)$$

Figure 1 : Problème de départ

Où nous avons :

- $f(C_t)$ qui représente le coût du cluster C_t ;
- T est l'ensemble des clusters t possibles ;
- y_t vaut 1 si t est dans la partition solution, 0 sinon ;
- a_{it} vaut 1 si l'objet i appartient au cluster t, 0 sinon ;
- p est le nombre de clusters désiré ;

La contrainte (1) assure le fait que nous ayons une partition, c'est à dire que chaque point se trouve dans un et un seul cluster. Le fait que la partition finale comporte p clusters se traduit par la contrainte (2), et la contrainte (3) signale que les variables y_t sont binaires.

1.2.2 Utilisation du Branch And Bound

La résolution d'un système linéaire en nombre entier, tel celui présenté en figure 1, est relativement difficile. La plupart des méthodes utilisent la forme relaxée du problème, c'est à dire permettent des solutions fractionnaires au problème. Ces solutions, plus faciles à trouvées, sont ensuite, si cela est possible,

passées sous forme entière.

La forme relaxée du problème de départ est la suivante :

$$min \sum_{t \in T} f(C_t)\, y_t \quad (R)$$

$$sous\, contraintes: \quad \sum_{t \in T} a_{it}\, y_t = 1 \quad (4)$$

$$\sum_{t \in T} y_t = p \quad (5)$$

$$y_t \in [0,1] \quad (6)$$

Figure 2 : problème relaxé

Les variables y_t ne sont plus binaires, mais continues sur l'intervalle [0,1].

Une solution envisagée pour résoudre le problème est une énumération des solutions entières possibles. Cela est réalisé à l'aide d'un arbre dont chaque branche représente une contrainte, et dont les feuilles représentent les solutions entières. Cependant, pour la majorité des problèmes, une telle énumérations est impensable, car le nombre de solutions possibles est trop élevé (de l'ordre de 2^n - 2 solutions, où n représente le nombre de points à classer).

Nous introduisons donc une méthode d'élagage de l'arbre, ce qui permet de réduire le nombre de solutions et d'accélérer la vitesse de résolution du problème.

Le parcours de l'arbre et l'élagage constituent la méthode de Branch And Bound, ou de séparation et d'évaluation progressive en français. Cette méthode fonctionne de la manière suivante : on relaxe et on résout le problème initial pour obtenir la racine de l'arbre. Cette solution n'est pas forcément entière, et on ajoute donc des contraintes de branchement pour obtenir de nouveaux problèmes linéaires relaxés. Ces contraintes sont expliquées plus en détails dans le document [5]. On résout ensuite les problèmes créés, on applique de nouveaux les contraintes sur les solutions obtenues, et ce jusqu'à obtenir un arbre dont toutes les feuilles sont des solutions entières ou des solutions fractionnaires de moins bonne qualité que la meilleure solution entière trouvée.

1.2.3 Méthode de résolution par génération de colonne

Nous l'avons vu le nombre de solutions possibles pour les problèmes de classification est exponentiel. Ces solutions seront notées pour la suite comme étant les colonnes de la matrice $A = (a_{it})_{.\leq i,t \leq n}$, avec n le nombre de points à classer. Le terme colonne, employé par la suite, fait référence aux colonnes de A, qui représentent les clusters.

La méthode de génération de colonne consiste à ne considérer non pas toutes les solutions possibles du problème, mais seulement un sous – ensemble de ces solutions. On cherche donc la solution optimale d'un problème, appelé problème maître, qui est identique au problème de départ mais qui contient seulement un nombre limité de colonnes. Une fois la solution optimale trouvée, un second

problème, appelé sous – problème ou problème auxiliaire, permet de chercher s'il existe une ou plusieurs colonnes améliorant la solution du problème initial et qui n'est pas comprise dans le problème maître. C'est donc l'algorithme qui résout le sous – problème qui constitue le générateur de colonne.

La génération de colonne fonctionne donc de la manière suivante :

Étape 1 : Résolution du problème maître jusqu'à l'optimalité ; récupération des variables duales ;
Étape 2 : Résolution du sous – problème pour trouver une colonne améliorante ;
Étape 3 : Si aucune colonne n'est trouvée, fin de l'algorithme ; sinon on ajoute la colonne au problème maître et on recommence à l'étape 1 ;

Figure 3 : La génération de colonne

La résolution du problème auxiliaire se fait de la manière suivante : il faut trouver une colonne C ayant un cout réduit négatif.

Le coût réduit d'une colonne est basé sur le vecteur dual associé au problème (P)[1]. Le vecteur dual est noté $(u, \mu) = (u_1, .. u_n, \mu)$.

Le coût réduit est calculé de la manière suivante :

$$\phi(C) = f(C) - \sum_{i=1}^{n} u_i a_i - \mu$$

Le sous – problème peut donc se résumer comme suit :

$$\exists ? C = (a_i)_{0 \le i \le n} \in \{0,1\}^n \; tel \; que \; \phi(C) < 0$$

Figure 4 : Problème auxiliaire

Ce problème peut être résolu de plusieurs manières, grâce à des méthodes heuristiques, des méthodes exactes, ou les deux, comme nous le verrons dans la partie 3 en page 19.

On obtient ainsi une colonne $(a_i)_{i=\{1..n\}}$ de la matrice A qui représente le cluster en variables binaires. Si le problème auxiliaire a une solution, alors celle – ci est ajoutée au problème maître, sinon on a résolu le problème (R)[2].

1.2.4 Le projet existant

Le but de mon stage, ainsi que de celui des autres stagiaires de cette année, était de travailler en équipe sur le projet ayant été commencé l'an dernier par les stagiaires de l'ISIMA. Lors de ce stage ils ont développé un programme permettant de faire de la classification non supervisée selon différents critères, tel par exemple la somme des diamètres, critère que j'expliquerai dans une partie

1 Voir figure 1 page 5
2 Voir figure 2 page 6

ultérieure.

Le programme comporte cinq grandes parties, qui sont la gestion du problème maître, la gestion des sous problèmes, l'interface avec les logiciels de programmation linéaire, le stockage des données et la gestion du Branch And Bound. Ces cinq parties sont articulées autours de la classe centrale *MasterProblem*, comme le montre la figure 5 en page suivante.

Une importante source de documentation que nous avons utilisée pour nous familiariser est les rapport de stage de l'équipe précédente ([2] à [6]).

La classe *MasterProblem* fait donc le lien entre toutes les parties du programme grâce à ses attributs et méthodes.

La classe *Column*, ainsi que ses dérivées spécifiques à chaque sous problème, représente les clusters. Cette classe étant une fille de la classe *BitVector*, les colonnes sont des vecteurs de bits, de taille le nombre d'observations. Elle comporte de plus un attribut « cost » pour stocker le coût du cluster. Les objets sont quant à eux des points, et leur présence dans un cluster est représentée par un 1 dans le vecteur, contre un 0 pour leur absence. De plus l'ordre des points est défini arbitrairement à celui de leur lecture dans le fichier d'entrée.

Les matrices (classes *Matrix* et *SymMatrix*) sont des vecteurs de vecteurs doubles servant à stocker les dissimilarités. Les dissimilarités sont des valeurs numériques positives permettant de mesurer les différences entre deux observations. La présence de deux classes distinctes vient du fait que les dissimilarités ne sont pas forcément symétriques, même si elles le sont pour la plupart des critères sur lesquels nous avons travaillé. Ces valeurs sont utilisées pour calculer le coût d'une colonne. Ainsi, pour le critère de la somme des diamètres, le coût d'une colonne est la plus grande dissimilarité entre deux points de cette colonne.

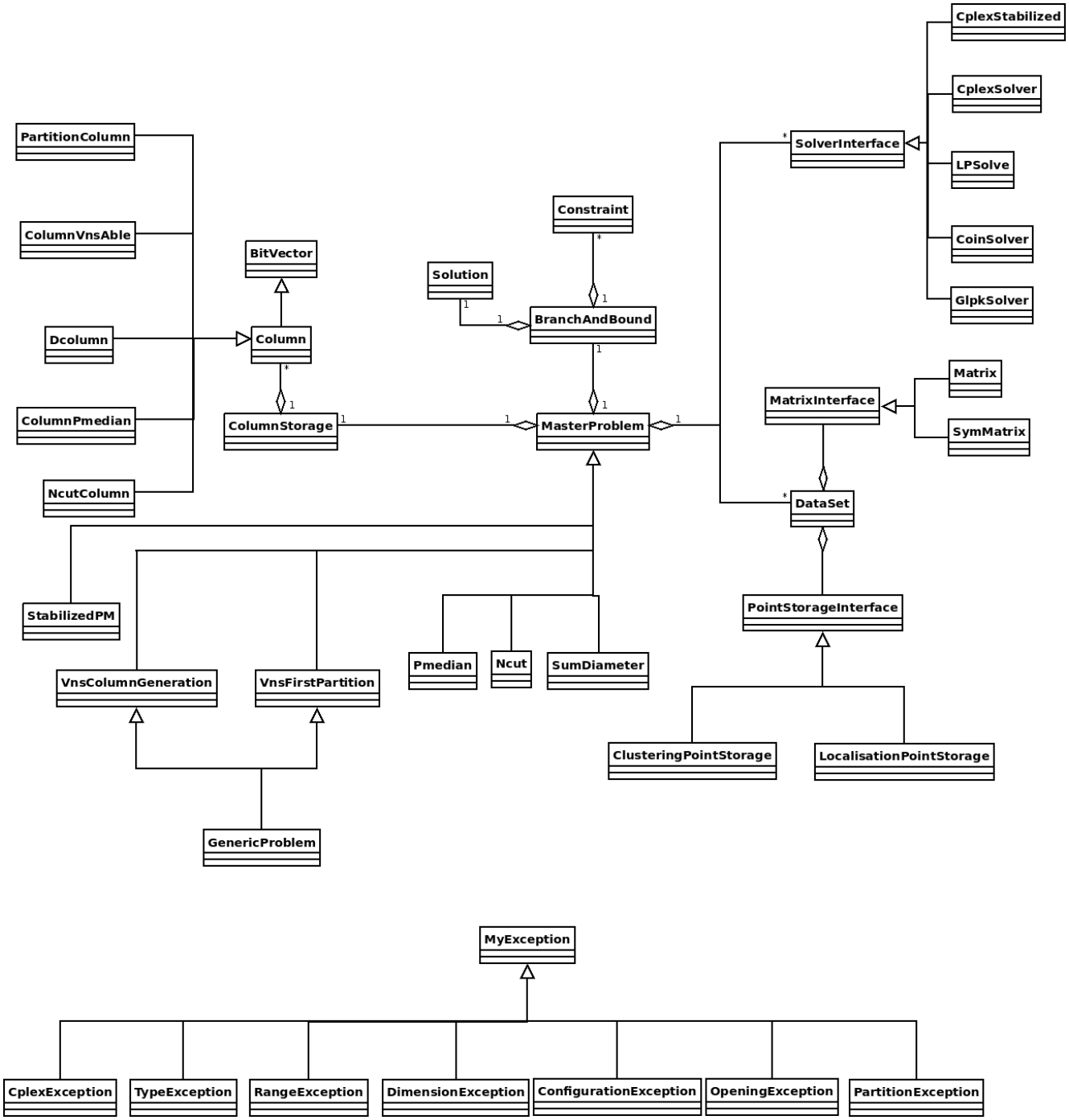

Figure 5 : Diagramme de classe UML de l'application

Deuxième partie

Gestion du code

2 Gestion du code

2.1 Passage de 32 à 64 bits

L'une des premières choses à faire sur le programme a consisté à le mettre à jour. En effet les serveurs qu'utilisaient les stagiaires de l'an passé, et donc pour lesquels avait été créé le programme, étaient en 32 bits. Une mise à jour ayant été effectuée, les serveurs sont maintenant passés en 64 bits. Il y a donc logiquement eu quelques changements à faire pour que le programme fonctionne de nouveau correctement.

Les principaux changements à effectuer se sont situés au niveau des librairies utilisées. En effet les libraires standards du C++ ne sont pas les mêmes en version 32 ou 64 bits. Il a donc fallu changer les inclusions des fichiers d'en-tête pour ces librairies là.

De même la librairie permettant d'utiliser le logiciel de programmation linéaire Cplex avec du C++ fut elle aussi à changer. De plus des modifications furent cette fois ci à faire dans le makefile.

2.2 Utilisation de la classe accuracy

L'un des points important du programme est la précision des calculs. En effet les erreurs d'arrondis, présentent dès lors que l'on fait de la programmation numérique, peuvent entrainer des erreurs importantes dans le programme, tel par exemple le cyclage d'une descente. Pour remédier à ça, les stagiaires de l'an dernier ont implémenté une classe *accuracy* qui permet de gérer la précision de la plus grande majorité des calculs effectués.

Cette classe permet de fixer la précision, actuellement à 10e-6, pour tout ce qui touche à la comparaison entre deux doubles : leur égalité, savoir lequel des deux est supérieur à l'autre, ou savoir si un double est nul ou non à la précision souhaitée. Elle porte sur des doubles car c'est le type de tous les attributs importants du programme, tel le coût d'une colonne ou les dissimilarités entre les points.

Cependant les stagiaires de l'an dernier n'avaient pas utilisé les méthodes de cette classe systématiquement. Cela créait donc des problèmes car certains calculs étaient effectués à une précision de 10e-6 et d'autres, ceux utilisant la librairie standard du C++, à 10e-12. Il a donc fallu parcourir le code pour faire en sorte que toutes les comparaisons de double se fassent grâce à la classe *accuracy*.

Cela à permis de régler certains problèmes dont nous n'arrivions pas à trouver la source et qui dépendaient en fait de la précision des calculs.

2.3 Main Global

A notre arrivée il n'existait pas de main global permettant de choisir « à la main » son critère. Chaque sous problème avait ainsi son main, et il fallait veiller à compiler le programme avec les bonnes options pour pouvoir avoir l'exécutable souhaité.

Par soucis de simplifier l'utilisation du programme en ligne de commande, et pour pouvoir plus facilement faire le lien avec l'interface graphique qu'Isabelle a développée cette année, nous avons décidé de créer un main global, permettant de choisir non seulement son critère mais aussi le type de solver que l'on veut utiliser (Cplex ou Glpk) et si on veut utiliser ou non la stabilisation, choses qui n'étaient pas réalisables avec les mains spécifiques à chaque critère. De plus la possibilité d'exporter les fichiers de sortie au format Latex ou Excel a été rajoutée suite à la création par Florian des classes le permettant.

Les mains des différents sous problèmes étant tous construits de la même façon (appel des méthodes *buildBranchAndBound* et *widthTreeCover*), la mise en commun n'a pas été compliquée. Le choix du critère se fait grâce à un entier passé en paramètre, qui se répercute ensuite par un changement d'un paramètre de la méthode *buildBranchAndBound*. Il en va de même pour le choix du solver et de l'utilisation ou non de la stabilisation.

La possibilité d'exporter les fichiers de sorties étant optionnelle, il a fallu trouver comment faire en sorte de prendre en compte ces options uniquement lorsqu'elles sont présentes. Il a donc été convenu de les placer à la fin de la ligne de commande, et les deux derniers arguments sont testés à chaque fois pour voir s'il s'agit ou non de ces options.

La ligne de commande finale du main global ressemble donc à ceci :

```
exeGlobal ProblemType Stabilization SolverType InputFile NbOfCluster
[NbOfClusterFinal] [OutputOptions]

ProblemType : 1 - Normalized Cut, 2 - Clique, 3 - Sum of Diameters,
5 - Sum of Stars
Stabilization : 1 - no, 2 - yes
SolverType : 1 - Cplex, 2 - Glpk
NbOfClusterFinal : if you wan't to do tests from nbOfCluster to
NbOfClusterFinal
OutputOptions = -e or -E to output results in CSV file
                -l or -L to output results in LaTeX file
```

Figure 6 : Ligne de commande du main global

2.4 Gprof

2.4.1 Description

Dans le but d'optimiser le code du programme, nous avons utilisé le logiciel Gprof. C'est un logiciel libre, faisant partie des utilitaires binaires GNU, et qui

permet d'effectuer du profilage de code, c'est à dire de créer une liste de toutes les méthodes appelées par une application et d'indiquer le temps passé dans celles ci. Cela permet ainsi de savoir quelles parties du code prennent le plus de temps et sont donc à optimiser en priorité, et quelles parties sont rapides et ne nécessitent donc pas de retouche.

Les fichiers de profilage générés sont présentés ainsi :

```
Flat profile:

Each sample counts as 0.01 seconds.
 %   cumulative   self              self     total
time   seconds   seconds    calls   s/call   s/call  name
18.02    250.58   250.58                             solveUright2
14.51    452.34   201.76    617220   0.00     0.00   SumDiameter::calculDiam(Dcolumn&)
10.63    600.20   147.86                             test_x
10.49    746.08   145.88 139478351   0.00     0.00   SumDiameter::entr(Dcolumn&, int)
9.72     881.20   135.12                             CPXPselectSFullNorm
4.70     946.62    65.42  961438177   0.00     0.00  SymMatrix::get(unsigned long const&,
unsigned long const&)
2.06     975.28    28.66                             CPXPrmatbld
1.84    1000.89    25.62  5256405473  0.00     0.00  DataSet::getDissimilarity(unsigned
long, unsigned long)
1.62    1023.46    22.57                             solveUright
1.34    1042.05    18.59                             upddnorm1
1.25    1059.46    17.41                             solveUrightSV
```

Figure 7 : Exemple d'un fichier de profilage généré par Gprof

2.4.2 Utilisation

Pour pouvoir utiliser Gprof il faut rajouter une option à la compilation des fichiers. Il a donc fallu modifier le makefile en conséquence, et une option PROFILING= -pg a ainsi été ajoutée.

L'utilisation de Gprof se fait ensuite simplement : il suffit de lancer l'application dont on souhaite faire le profilage, de la laisser se terminer normalement, puis de lancer Gprof avec le fichier gnom.out que l'application a créé. Cependant l'application connait certains ralentissements dus à l'utilisation de Gprof, le temps d'exécution pouvant parfois être doublé.

Dans le cadre de mon stage j'ai donc été amené à utiliser Gprof pour optimiser le code du critère de la somme des diamètres. Après avoir effectué plusieurs profilages sur des problèmes différents, il est apparu que les méthodes qui prenaient le plus de temps étaient, outre les méthodes de résolution du solver, la méthode de calcul du diamètre et celle faisant la mise à jour du diamètre d'une colonne lors de l'entrée d'un point dans celle-ci. De plus la méthode *getDissimilarity*, qui permet d'aller chercher une valeur dans la matrice de dissimilarités, était appelée un très grand nombre de fois.

L'utilisation de Gprof m'a ainsi permis de savoir sur quelles méthodes il était le plus utile d'apporter des modifications, et donc de ne pas perdre de temps en optimisant d'autres méthodes « inutilement ».

L'optimisation de ces méthodes sera développée dans la partie suivante.

2.5 Création de classes

2.5.1 Création de classes mères pour les solver

Afin de rendre le code un peu plus lisible, il a été convenu que des classes mères *CplexInterface* et *GlpkInterfaces* seraient créées. En effet dans le programme existant étaient présentes deux classes *CplexSolver* et *CplexSolverStabilized*. De même une classe *GlpkSolver* existait et j'ai créé une classe *GlpkSolverStabilized* sur le même principe. Ces classes, qui comme leur nom l'indique permettent de faire le lien avec les logiciels de programmation linéaire, regroupaient cependant des méthodes identiques, et créaient ainsi une redondance de code inutile.

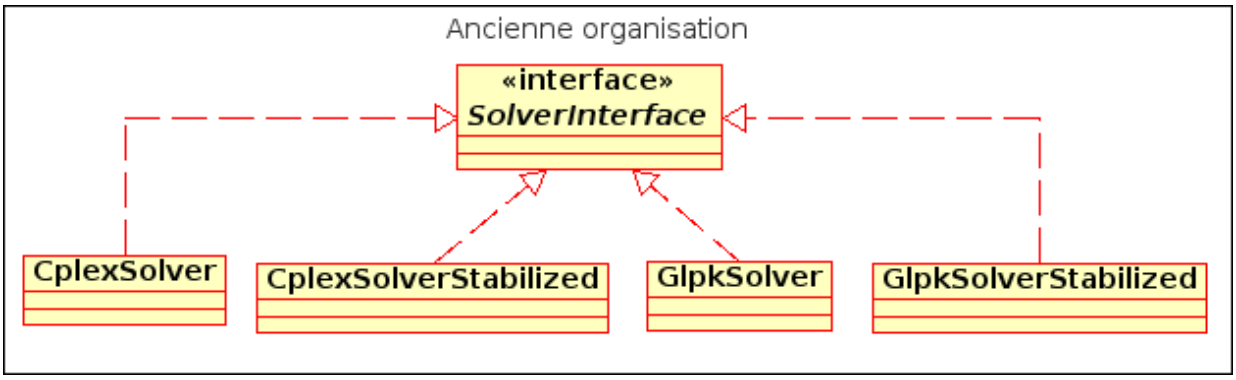

Figure 8 : Ancien diagramme de classe UML de la partie solver

Pour réduire cette redondance, j'ai créé des classes intermédiaires GlpkInterface et CplexInterface, tel que représenté sur la figure 5, regroupant toutes les méthodes identiques présentes dans les classes en dérivant, tels le constructeur, les méthodes d'écriture dans un fichier, ou encore celle de fermeture du programme.

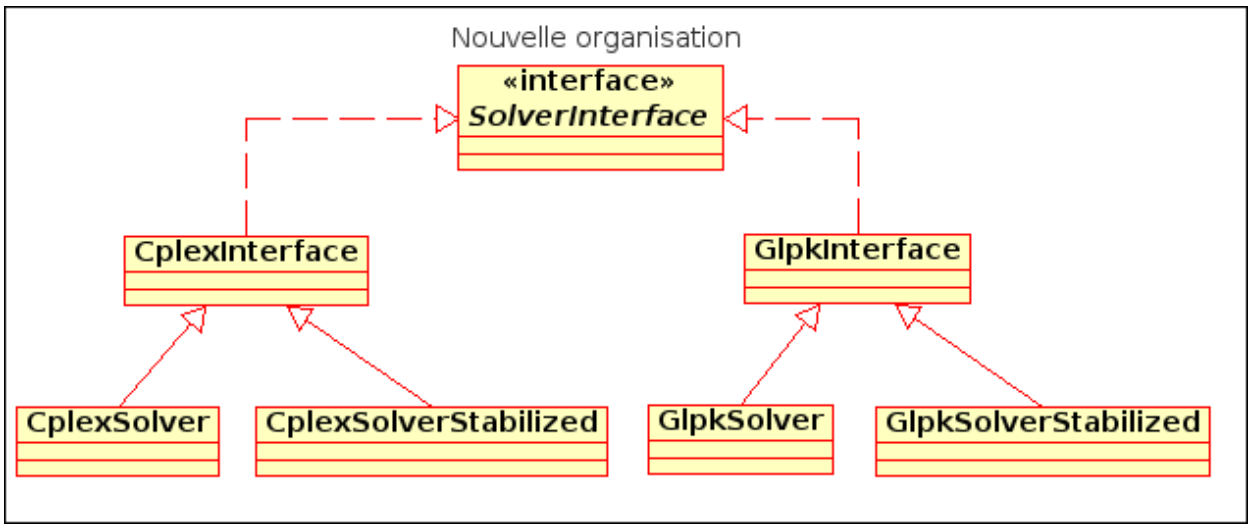

Figure 9 : Nouveau diagramme de classe UML de la partie solver

Notons qu'il a fallu faire quelques modifications dans le code suite à ce changement. En effet Cplex étant un logiciel propriétaire, son utilisation nécessite une licence, qui coûte relativement cher. Or pour le critère de la somme des diamètres, la méthode exacte implémentée se sert du solver Cplex. Les classes *CplexSolver* et *CplexSolverStabilized* avaient donc été créées pour que la même licence soit utilisée dans la résolution de la méthode exacte de la somme des diamètres et dans la résolution du problème maître, et il a fallu modifier quelque peu les classes associées à Cplex pour que cela soit toujours le cas.

2.5.2 Création de la classe Example

Le programme sur lequel porte mon stage a été commencé l'an dernier et sa conception se poursuivra encore au moins l'an prochain. Ainsi dans l'objectif de laisser un programme sur lequel il sera facile de revenir, il a été décidé de créer une classe servant d'exemple pour l'éventuel ajout d'un futur critère. Cela a été facilité par le fait que la plupart des critères actuels sont codés de façon sensiblement identique.

Une telle classe se doit donc d'être particulièrement bien commentée, et doit comporter toutes les méthodes indispensables pour qu'un critère puisse être ajouté.

De plus une implémentation de ces méthodes fut réalisée, en mettant à chaque fois les différentes possibilités offertes, telle l'utilisation de l'heuristique de recherche à voisinage variable (Variable Neighborhood Search en anglais)[3] générique ou quadratique, toute deux présentes dans le programme.

Afin de faciliter la tâche de programmeurs voulant reprendre le projet, Isabelle Barclay et moi – même avons réalisé un tutoriel, présent en annexe du rapport, expliquant les diverses manipulations à effectuer pour l'ajout d'un critère, telles les modifications à apporter au makefile.

3 Voir description complète dans la section 3.2 en page 24

```
// If you use the generic VNS, you have to create your class like this :
class Example : public VnsColumnGeneration, public VnsFirstPartition
{
	protected:
			// place here the attributs

	public:

	/// Constructor
	Example(DataSet * inDataSet, SolverInterface * inSolver) :
		MasterProblem(inDataSet, inSolver)
		{
			// initialisation of the random generator
			long int graine = time(NULL);

			// loop in order to make srand working better
			srand(graine);
			int i = rand()%1000000;
			std::cout << "graine : " << graine << std::endl;
			for (int k =0; k<i;k++)
			{
				lrand48();
			}
		}


	/// Destructor
	 ~Example(){}

	/// \brief Compute the cost of a column
	/// \param col The column we want to compute the cost of
	/// \return The cost of col
	double evalCost(const Column & col);

	/// \brief Generate columns and add them to the problem.
	/// \param NbCol Number of column to add to the problem.
	/// \return Total number of generated columns.
	uint heuristicCreateColumns(uint nbCol);

	/// \brief Generate columns for the master problem
	/// \param NbCol number of columns you want
	/// \return Number of created columns
	uint createColumns( uint NbCol);

	/// \brief Launch the exact method
	/// \param nbCol the number of column wanted (=1)
	/// \return Number of created columns
	uint exactCreateColumns(uint nbCol);

	/// \brief Create a first partition (with VNS)
	/// \return Number of created columns (always nbObs)
	uint createFirstColumns();

};
```

Figure 10 : En-tête de la classe Example

2.5.3 Utilisation d'une partition

Initialement le programme prend en entrée un fichier de points, en calcule les dissimilarités, puis lance l'heuristique initiale et le reste du programme.

Une méthode l*oadFirstPartition* permet de lire une partition dans un fichier, ce qui permet de tester si la seconde heuristique et la méthode exacte fonctionnent correctement, par exemple en fournissant directement la solution optimale au programme. Cependant cette méthode n'était pas commentée, et nous n'avions donc aucune idée de comment organiser les données dans le fichier qu'elle lit, même après examen du code correspondant.

Heureusement une méthode *saveFirstPartition* était aussi présente dans le code. Cette méthode crée un fichier contenant la partition initiale dans le format voulu, et nous a donc permis de nous servir de la méthode l*oadFirstPartition*.

Il n'existait par contre aucune méthode permettant de sauvegarder la partition finale, et il fallait créer à la main le fichier la contenant, ce qui peut s'avérer fastidieux pour des problèmes de plusieurs centaines de points. J'ai donc créé une méthode, *saveOptimalPartition*, permettant de sauvegarder la partition optimale dans un fichier lisible par la méthode *loadFirstPartition*. Elle est basée sur le même principe que la méthode *saveFirstPartition*.

Troisième partie

Le somme des diamètres

3 La somme des diamètres

Le critère sur lequel j'ai été amené à travailler est celui de la somme des diamètres.

Le diamètre d'un cluster C est la plus grande dissimilarité entre deux points x et y présents dans ce cluster :

$$Diam(C) = max(d(x,y)), \forall (x,y) \in C^2$$

En voici une illustration :

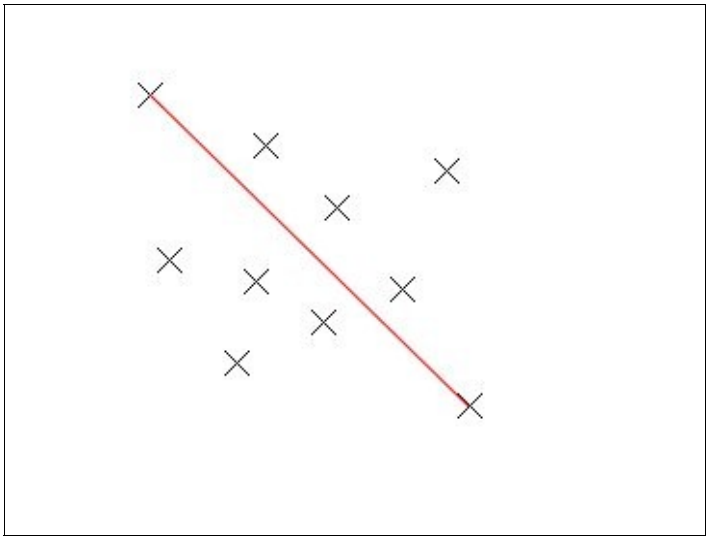

Figure 11 : Diamètre d'un cluster

La somme des diamètres est un critère d'homogénéité qui consiste à répartir n points en p clusters de telle sorte que la somme des diamètres des clusters formant la partition finale soit la plus petite possible, avec évidemment n > p.

Le problème mathématique associé à ce critère peut donc se formuler de la manière suivante, avec :

- T est l'ensemble des clusters t possibles ;
- y_t vaut 1 si t est dans la partition solution, 0 sinon ;
- a_{it} vaut 1 si l'objet i appartient au cluster t, 0 sinon ;

$$min \sum_{t \in T} Diam(C_t)\, y_t \quad (P)$$

$$sous\, contraintes: \quad \sum_{t \in T} a_{it}\, y_t = ١ \quad (1)$$

$$\sum_{t \in T} y_t = p \quad (٢)$$

$$y_t \in \{0,1\} \quad (3)$$

Figure 12 : Problème de la somme des diamètres

Les contraintes assurent que chaque point doit se trouver dans un seul cluster (contrainte (1)), que le nombre de cluster final est p (contrainte (2)), et enfin que les variables y_t sont binaires (contrainte (3)).

La gestion de ce critère a été implémentée et ajoutée au programme par les stagiaires de l'an passé, et mon travail consistait à apporter d'éventuelles améliorations. Pour cela il a fallu dans un premier temps que je parcours la littérature, pour mieux connaitre ce critère.

3.1 Modifications du code existant

3.1.1 Optimisation de méthodes

Comme nous l'avons vu dans la partie 2.4.2 sur l'utilisation du logiciel Gprof, un gain de temps et de performances du programme pour le critère de la somme des diamètres passe par une optimisation de méthodes, notamment celle qui permet de calculer le diamètre d'une colonne et celle qui calcule le coût de l'entrée d'un point dans une colonne. Je me suis donc penché sur, entre autre, ces deux méthodes.

La méthode *calculDiam*, qui prend en paramètre la colonne dont on veut calculer le coût, fonctionne selon l'algorithme suivant :

```
temp = -1
Pour chaque point1
  si le point1 est dans la colonne
    pour chaque point2 supérieur à point1
        si point2 est dans la colonne
        et si la dissimilarité entre point2 et point1 est supérieure à temp
              temp = la dissimilarité entre point2 et point 1
              point1 et point2 forment le diamètre de la colonne
          fin si
     fin pour
fin pour
on met à jour le coût de la colonne
```

Figure 13 : Algorithme initial de la méthode calculDiam

Pour connaître la dissimilarité entre les points 1 et 2 on fait un appel à la méthode *getDissimilarity*. On fait ainsi deux appels à cette méthode si le point2 est dans la colonne, alors qu'on pourrait n'en faire qu'un seul et stocker le résultat dans une variable. Cela ferait faire moins d'appels à la méthode et donc gagner, en théorie, un peu de temps.

J'ai effectué ce changement, et fait en sorte que l'appel à la méthode *getDissimilarity* ne soit effectué que lorsque les deux points sont présents dans la colonne. Le nouvel algorithme est le suivant :

```
Temp = -1
diss  = 0
Pour chaque point1
  si le point1 est dans la colonne
    pour chaque point2 supérieur à point1
      si point2 est dans la colonne
          diss =  getDissimilarity(point1, point2)
          si diss > temp
            temp = diss
            point1 et point2 forment le diamètre de la colonne
        fin si
    fin pour
fin pour
on met à jour le coût de la colonne
```

Figure 14 : Nouvel algorithme de la méthode calculDiam

De plus l'ancien algorithme était utilisé à plusieurs endroits dans le code pour calculer le diamètre d'une colonne, notamment dans la méthode calculant le coût d'entrée d'un point, qui prenait elle aussi du temps. J'ai donc effectué ce changement dans toute les méthodes où cet algorithme apparaissait, et ai lancé des tests pour savoir si on gagnait effectivement du temps ou pas.

Cette optimisation, ainsi que les quelques autres que j'ai apporté, ont chacune contribué à accélérer le programme, même si aucune d'entre elle n'apporte de changements réellement significatif.

3.1.2 Modification de méthodes

En plus d'optimiser le code, il a aussi fallu effectuer des changements dans certaines méthodes. Ainsi la méthode *FirstLocalSearch*, qui est utilisée dans l'heuristique, doit, comme son nom l'indique, retourner le premier optimum local trouvé. Or, en regardant le code je me suis rendu compte qu'elle ne retournait non pas le premier optimum, mais le meilleur, devenant ainsi identique, dans le principe du moins, à la méthode *BestLocalSearch*, implémentée elle aussi. J'ai donc modifié cette méthode pour qu'elle fasse vraiment ce qu'on attend d'elle.

Ce changement n'a cependant pas modifié les résultats obtenus, car la méthode *FirstLocalSearch* n'est appelée qu'au tout début du programme. Cela à cependant fait gagner quelques secondes à l'exécution.

La méthode *CreateFirstColumn*, qui permet, comme son nom l'indique, de créer les premières colonnes fournies au programme, semblait être une version encore en construction. Il y avait en effet plusieurs tests et variables inutiles présents dans le code. En voici l'algorithme :

```
i = 0
value = 1.000.000
tant que i < 1
    créer les premières colonnes
    et les stocker dans la partition courante

    si la partition courante a un coût inférieur à value
       la partition courante devient la meilleur partition
       value reçoit le coût de la partition
    fin si
    i = i + 1

    si i = 1
        ajouter la meilleur partition au problème maître
    fin si
fin tant que
complèter la partition
```

Figure 15 : Algorithme initial de la méthode CreateFirstColumn

Remarquant que la boucle tant que i est inférieur à 1 était inutile, de même que le test pour savoir si i est égal à 1, j'ai enlevé le test et est changé la condition d'arrêt de la boucle tant que, en remplaçant le 1 par 10. Cette nouvelle version permet plus de possibilités d'amélioration et semble être l'idée originale. L'algorithme pensé était donc probablement le suivant :

```
i = 0
value = 1.000.000
tant que i < 10
    créer les premières colonnes
    et les stocker dans la partition courante

    si la partition courante a un coût inférieur à value
       la partition courante devient la meilleur partition
       value reçoit le coût de la partition courante
    fin si
i = i + 1
fin tant que

ajouter la meilleur partition au problème maître
complèter la partition
```

Figure 16 : Algorithme temporaire de la méthode CreateFirstColumn

La valeur 10 a été choisie arbitrairement, et d'autres valeurs ont aussi été

testées. Les tests effectués avec cet algorithme ont donné des résultats décevants : les temps d'exécution étaient doublés pour des résultats identiques, et ce quelque soit le nombre d'essais laissés. On est donc revenu à l'algorithme de départ, dans une écriture simplifiée comme suit :

```
on créer les premières colonnes
et les stocker dans la partition courante

si la partition courante a un coût inférieur à 1.000.000
    la partition courante devient la meilleur partition
    value reçoit le coût de la partition courante
fin si

ajouter la meilleur partition au problème maître
complèter la partition
```

Figure 17 : Algorithme final de la méthode CreateFirstColumn

3.1.3 Changement de type de contraintes du problème maitre

On a vu à la figure 12, que la fonction objectif est soumise à des contraintes permettant d'assurer le fait qu'on ai une partition, et que celle ci soit du bon nombre de clusters.

Cependant dans le cas de la somme des diamètres, la contrainte qui fait que chaque point doit se trouver dans un seul cluster, à savoir :

$$\sum_{t \in T} a_{it} y_t = 1$$

peut être changée en la contrainte suivante :

$$\sum_{t \in T} a_{it} y_t \geq 1$$

c'est à dire qu'un point peut être dans plusieurs clusters.

Cela peut paraitre surprenant, car on veut une partition des points et on permet cependant à un point d'être dans plusieurs clusters. La particularité de la somme des diamètres vient du fait qu'un point peut changer de cluster plusieurs fois sans que cela change le diamètre des clusters concernés et donc la fonction objectif. Le changement de type de contrainte fait qu'elles sont ainsi plus rapidement respectées, et devrait donc permettre un gain de temps relativement important.

De plus on est quand même assuré d'avoir une partition car le programme ne met effectivement le point que dans une seule colonne. Le changement de type de contraintes permet seulement au solver de trouver plus facilement une solution réalisable du problème, lui évitant de devoir calculer toutes les colonnes possibles pour un point.

La possibilité de pouvoir faire ce changement de type de contraintes a amenée à modifier quelque peu les classes *GlpkSolver* et *GlpkSolverStabilized*, tandis que

les classes *CplexSolver* et *CplexSolverStabilized* étaient quand à elles prévues pour cet éventuel changement.

Une fois ce changement effectué, j'ai lancé des tests pour voir si on avait effectivement un gain de temps ou non. Ces tests ont été effectués sur un fichier de 59 points. Les temps sont exprimés en secondes et ceux affichés sont la moyenne des temps de dix tests. Le terme « contraintes de type E » signifie que le programme a été exécuté avec la contrainte suivante :

$$\sum_{t \in T} a_{it} y_t = 1$$

et le terme « contraintes de type G » signifie que le programme a été exécuté avec la contrainte suivante :

$$\sum_{t \in T} a_{it} y_t \geq 1$$

Nombre de cluster	Temps total moyen avec des contraintes de type E	Temps total moyen avec des contraintes de type G	Intervalle de temps avec contraintes de type E	Intervalle de temps avec contraintes de type G
3	726.419	194.88	[80.17 - 3155.25]	[28.17 - 615.09]
4	73.816	71.387	[45.16 - 111.26]	[14.4 - 167.97]
5	147.615	70.407	[97.93 - 269.26]	[30.13 - 152.59]
6	443.634	303.532	[368.46 - 715.84]	[157.41 - 752.1]
7	388.372	163.135	[225.53 - 613.16]	[23.64 - 579.38]
8	732.431	281.786	[217.9 - 1079.61]	[132.9 - 459.93]

Figure 18 : Comparaison de résultats

On remarque ainsi que cette modification apporte réellement un gain de temps important. Cependant de plus amples tests seront à mener pour voir si ce gain est présent pour tous les types de problèmes, qu'ils soient difficiles ou simples. Il va de soit que les résultats obtenus concernant la fonction objectif restent les mêmes avant et après le changement.

3.2 Nouvelle heuristique de départ : la V.N.S

Dans le but d'améliorer le critère de la somme des diamètres nous avons testé une nouvelle heuristique de départ, c'est à dire une nouvelle manière de créer de façon stochastique les premières colonnes qu'on fournit au programme.

L'heuristique de départ existante consistait à créer p – 1 clusters contenant chacun 1 point, et 1 cluster contenant n – p + 1 points, de telle sorte que la colonne contentant n – p + 1 points soit de diamètre le plus petit possible. Cette méthode donnait des résultats mitigés, trouvant parfois la solution optimale, mais trouvant la plupart du temps une solution de l'ordre de 10% de la solution optimale, alors que les heuristiques initiales des autres critères implémentés dans le programme trouvent, dans le pire des cas, des solutions de l'ordre de 2% de la solution optimale. Nous avons décidé d'implémenter une V.N.S spécifique à la somme des diamètres.

Une V.N.S, pour « Variable Neighborhood Search », ou Recherche à Voisinage Variable, est un algorithme d'optimisation stochastique proposé en 1997 par Pierre Hansen et Nenad Mladenović. Le principe est de modifier progressivement une solution déjà existante du problème pour en trouver une meilleure.

Dans le cas d'une V.N.S spécifique à la somme des diamètres, les modifications se font en déplaçant un point formant le diamètre d'une colonne, appelé index, dans une autre colonne. L'algorithme de base de la méthode *descent*, qui effectue ces modifications, est donc le suivant :

```
trouvé = vrai
optimal = faux
Tant que trouvé
    on prend un point au hasard dans la liste des index
    on stock le coût de sa colonne1 et les index de cette colonne
    on choisi une colonne2 différente de colonne1
    on stock le coût de cette colonne et ses index
    on change le point de colonne
    on calcul les nouveaux diamètres
    si la somme des 2 nouveaux diamètres est inférieure à celle des 2 anciens
    ou si elle est égale
            on met à jour la liste d'index
            trouvé = vrai
            optimal = vrai
    sinon
            on remet les index et coût initiaux des colonnes
            on remet le point dans son ancienne colonne
            on change de colonne2
     fin si
fin tant que
retourner optimal
```

Figure 19 : Algorithme de la descente

Cette méthode rend vrai si elle a réussi à trouver une partition améliorante par rapport à celle qu'elle a reçue en paramètre, et faux sinon.

Pour choisir au hasard le point dans la liste, je me suis servis du générateur de nombres pseudo aléatoire *lrand48.* Par soucis de simplicité, la liste des index est un vecteur de pair contenant l'index et un pointeur vers la colonne à laquelle il appartient.

L'algorithme de la V.N.S en elle même est le suivant :

```
init (currentpartition)
Tant que temps < tempsMax et nbiter < nbiterMax
    on fait une descente sur currentpartition
    si la descentre trouve une partition améliorante
            on la stock dans bestpartition
            iter = 0
            magnitude = 0
    sinon
            shake(magnitude)
            on stock bestpartition dans currentpartition
            iter = iter + 1
            magnitude = magnitude + 1
    fin si
fin tant que
```

Figure 20 : Algorithme de la V.N.S

La méthode *shake* se contente de bouger magnitude index sans se soucier d'une amélioration de la solution, et est donc basée sur le même principe que la descente, avec le test en moins.

La méthode d'initialisation *init* est celle qui a posé le plus de problèmes. Il a en effet fallu trouver une méthode qui donne une solution réalisable, donc une partition ou une couverture des points, qui ne prend pas trop de temps, et qui a une part d'aléatoire. J'ai donc implémenté et testé plusieurs méthodes.

Tout d'abord j'ai mis l'ancienne heuristique de départ comme méthode d'initialisation. On est ainsi sur d'avoir une solution réalisable en un temps court et avec une part d'aléatoire. De plus elle présentait l'avantage d'être déjà implémentée. Cependant je me suis rendu compte au bout de quelques tests que ce n'était peut être pas la meilleur solution car le temps du programme étaient beaucoup plus long qu'avec l'ancienne version.

J'ai donc implémenté une méthode créant une partition des points, selon l'algorithme suivant :

```
cpt = 0
tant que cpt < n
    on se place au début de la partition

    tant que on est pas à la fin de la partition
       on choisi un point au hasard dans la liste des points

       si ce point n'a pas déjà été choisi
          on le met dans la colonne
          on passe à la colonne suivante
          on marque le point comme étant marqué
          on incrémente cpt
       fin si

       si cpt = n
          on va à la fin de la liste de la partition
       fin si
     fin tant que
fin tant que
```

Figure 21 : Algorithme de la méthode init

La liste de points est un vecteur dans lequel on met l'indice du point s'il n'a pas encore été choisi, et -1 dès que le point est placé dans une colonne. Grâce à cette méthode on est sûr d'avoir une partition des points et une part importante d'aléatoire. J'ai donc effectué des tests qui cette fois ci se sont avérés un peu plus concluant. En effet les temps sont voisins de ceux de l'ancienne heuristique. Ceci est donc encourageant.

Néanmoins la solution trouvée par la V.N.S est en général aux alentours de 85% de la solution optimale, contre environ 10% pour l'ancienne heuristique. Cependant le temps maximum laissé à la V.N.S, ainsi que le nombre d'itération maximum sans améliorations, ont été arbitrairement fixé à 0.33 secondes et 10 itérations. Les résultats pourront donc peut-être être améliorés en changeant ces paramètres. Mais je n'ai, au moment d'écrire ce rapport, pas encore eu le temps de faire des tests avec d'autres paramètres.

3.3 Méthode exacte de résolution du problème auxiliaire

Le but du programme est de trouver la solution optimale à coup sur pour chaque problème qu'on lui soumet. Pour cela il est décomposé en trois étapes : tout d'abord l'heuristique initiale crée rapidement une solution, pas forcément optimale comme nous l'avons vu. Ensuite une seconde heuristique cherche la solution optimale. Seulement celle-ci n'est pas sur de la trouver à tous les coups. Il faut donc une troisième étape, qui est l'utilisation d'une méthode exacte. Cette méthode doit pouvoir générer une solution meilleure que celle trouvée par l'heuristique si celle – ci n'est pas la solution optimale.

Idéalement la méthode exacte n'est appelée qu'une seule fois, pour vérifier que la solution trouvée par l'heuristique est la solution optimale. En effet

l'utilisation de la méthode exacte ralentie considérablement le programme (c'est d'ailleurs pour cela qu'on utilise une heuristique). Mais en pratique la méthode heuristique ne trouve pas toujours la solution optimale du premier coup, et il faut donc lancer plusieurs fois la méthode exacte.

Pour le critère de la somme des diamètres la méthode exacte, inspirée par Sylvain Perron, est un problème linéaire qui fait appel directement au solver. Si la résolution de ce problème fournie une colonne améliorante alors on l'ajoute au problème maître et on relance l'heuristique. Sinon l'optimalité a été atteinte.

La formulation du problème linéaire à résoudre est la suivante :

$$\begin{array}{lll} min & D_{max} - \sum u_i y_i - \mu & \\ s.c. & d(i,j)\, y_{ij} \leq D_{max} & \forall (i,j) \\ et & y_i + y_j \leq y_{ij} + 1 & \forall (i,j) \end{array}$$

Figure 22 : Problème linéaire associé à la méthode exacte

où :

- D_{max} est le diamètre de la colonne ;
- y_i vaut 1 si le point i est dans la colonne, 0 sinon ;
- y_{ij} vaut 1 si les points i et j appartiennent à la colonne, 0 sinon ;
- $(u_1, .., u_n, \mu)$ est le vecteur dual associé au problème (P) défini figure 12 ;

3.3.1 Passage à GLPK

Comme dit précédemment cette méthode fait directement appel au solver. Or l'an dernier elle n'a été implémentée que pour être utilisée par le logiciel Cplex. Comme nous allons le voir dans la partie suivante, j'ai été durant mon stage en charge de modifier le programme pour qu'il puisse être utilisé avec un autre solver, Glpk. Il a donc fallu que je modifie la méthode exacte de la somme des diamètres pour que celle – ci s'adapte dans un premier temps au solver utilisé, et, dans un deuxième temps, qu'elle puisse être utilisée avec Glpk.

Le code de la méthode *exactCreateColumns* était le suivant :

```
uint SumDiameter::exactCreateColumns(uint )
{
   uint added      = 0;

   added = !exactSol();

   return added;
}
```

Figure 23 : Ancien code de la méthode exacte

J'ai ainsi modifié cette méthode pour qu'elle appelle, suivant le solver utilisé, la bonne méthode. Pour cela il a fallu que j'ajoute un attribut dans les classes concernant les solvers qui permet au sous problème de savoir quel solver est utilisé par le programme, chose qui n'était pas possible avant. Le nouveau code de la méthode est donc le suivant :

```
uint SumDiameter::exactCreateColumns(uint )
{
   uint added      = 0;
   int  solverType = getSolverType();

   if(solverType == 1 || solverType == 2)
   {
      added = !exactSolCplex();
   }
   else if(solverType == 3 || solverType == 4)
   {
      added = !exactSolGlpk();
   }

   return added;
}
```

Figure 24 : Nouveau code de la méthode exacte

La méthode *exactSolCplex* est l'ancienne méthode *exactSol*. Celle ci crée un nouveau problème Cplex, qui est celui présenté figure 22, et le résout en faisant directement appel aux méthodes de la librairie permettant d'utiliser Cplex en C++. Elle ne passe pas par les méthode de la classe CplexSolver. J'ai donc créer la méthode *exactSolGlpk* qui fait la même chose mais en utilisant la librairie permettant d'utiliser Glpk en C++.

La modification de la méthode *exactSol* pour faire en sorte d'utiliser les méthodes des classes CplexSolver et GlpkSolver était impossible, car le problème résolu par la méthode exacte utilise des variables binaires. Or le problème maître et les sous - problèmes n'utilisent que des variables continues, et les méthodes des classes *CplexSolver* et *GlpkSolver*, qui ont été créées pour résoudre ces problèmes, ne sont ainsi prévues que pour utiliser des variables continues.

Une modification de la méthode exacte aurait donc entrainer des modifications importantes dans les classes des solvers et aurait été plus fastidieuse que de créer une nouvelle méthode.

3.3.2 Résultats

J'ai ensuite lancé des tests pour vérifier si la méthode marchait, et pour comparé les temps d'exécution avec les deux solvers. Je me suis ainsi rendu compte que Glpk affichait beaucoup d'informations inutiles au cours de la résolution du problème, et ai modifié ce problème.

Les résultats obtenus ont été mitigés. En effet la méthode exacte sous Glpk fonctionne correctement, elle trouve les mêmes résultats que Cplex, et dans des

temps sensiblement identiques pour des petits problèmes (jusqu'à 30 points). Cependant les temps d'exécution augmentent avec le nombre de points, mais de manière beaucoup plus importante que pour Cplex. Ainsi pour un problème de 60 points, là où le programme met en moyenne 30 secondes pour résoudre le problème avec Cplex, il met jusqu'à 80.000 secondes avec Glpk.

On s'est ainsi rendu compte que Glpk n'était pas un bon concurrent pour Cplex sur les problèmes linéaires en variables binaires, et qu'il faudrait continuer d'utiliser Cplex pour le critère de la somme des diamètres.

Quatrième partie

GLPK

4 Glpk

4.1 Description

Comme nous l'avons vu précédemment, le programme utilise des logiciels de programmation linéaire, et plus particulièrement Cplex. Or ce dernier est un logiciel propriétaire, créé par l'entreprise Française ILOG, dont les licences sont relativement onéreuses, et le GERAD n'en possède que quelques unes. L'utilisation du programme est donc souvent limitée par la disponibilité ou non d'une licence Cplex.

Pour remédier à ce problème Gilles et Sylvain ont voulu utiliser le logiciel libre Glpk (GNU Linear Programming Kit). Les stagiaires de l'an dernier ont donc commencé à implémenter, en même temps que pour Cplex, les classes permettant au programme de se servir de Glpk. Seulement cette idée a été abandonnée pour se concentrer sur Cplex, qui est à priori plus sur et plus performant.

Cette année cependant nous avons décidé de terminer cette partie. En effet, maintenant que le programme fonctionne correctement avec Cplex, il semble intéressant de pouvoir comparer les performances des deux solvers, et d'étudier ainsi la possibilité de ne plus utiliser que Glpk à la place de Cplex. Cette décision a aussi été prise dans le but de mettre le programme en cours à disposition du grand public une fois terminé. L'utilisation d'un solver gratuit rendrait ainsi plus pratique celle du programme, et ce pour plus de personnes.

Glpk est un ensemble de fonctions écrites en C, organisées sous forme d'une librairie, qui permettent de résoudre entre autre des problèmes linéaires (en variables continues) et des problèmes linéaire en nombres entiers. Il est, de part sa conception, relativement simple à utiliser en C et dans les langages en dérivant, tel le C++.

J'ai donc commencé par lire le manuel d'utilisation de Glpk, présent sur le site internet du logiciel ([7]) et par chercher un document me permettant de faire le lien entre les fonctions de Cplex et celles de Glpk ([8]). Ceci m'a été très utile car les classes pour Cplex existaient déjà dans le programme. J'ai ainsi pu vérifier plus facilement si les méthodes des classes pour Glpk étaient correctes en les comparant à celles des classes pour Cplex.

4.2 Implémentation du solver

Pour ajouter au programme la possibilité d'utiliser Glpk, il m'a fallu finir la classe *GlpkSolver* et créer les classes *GlpkSolverStabilized* et, comme nous l'avons vu précédemment, *GlpkInterface*. La présence de deux classes distinctes vient du fait que le solver peut être utilisé de deux façons : avec ou sans stabilisation.

La stabilisation est une méthode permettant au programme de cibler les recherches à effectuer, le rendant ainsi théoriquement plus rapide. Le choix d'utiliser ou non la stabilisation est fait par l'utilisateur au moment de lancer le programme. Un problème stabilisé contenant plus d'informations qu'un problème « normal », telles des bornes supplémentaires, ne sera pas fourni de la même manière au solver, et nécessite donc une classe différente.

4.2.1 Sans stabilisation

La classe GlpkSolver avait été débutée par les stagiaires de l'an passée et contenait déjà les méthodes les plus importantes, à savoir celles permettant de résoudre le problème, les méthodes servant à ajouter des colonnes au problème, ou encore celle permettant au solver de retourner la solution au programme. J'ai donc dans un premier temps vérifier si ces méthodes étaient finies et faisaient ce qu'il fallait.

J'ai ensuite dû créer les méthodes nécessaires à l'utilisation de Glpk, telle la méthode d'initialisation du solver, qui se charge de transcrire le problème fourni par le programme en problème compréhensible par le solver, ou les méthodes permettant de modifier les contraintes appliquées aux colonnes et aux points les composants.

L'une des parties difficiles de cette implémentation à été de faire le lien entre les différents index que prennent les colonnes au cours du programme. En effet les colonnes sont créées avec un index par le solver, puis envoyées au problème secondaire, qui vérifie si elles sont améliorantes ou non, et qui, le cas échéant, les fait parvenir jusqu'au problème maître qui les stocke. Cependant le solver est appelé plusieurs fois par le programme, et créé un nouveau problème à chaque appel. Une colonne fournie par le solver peut ainsi avoir le même index qu'une colonne déjà stockée par le problème maître.

Pour remédier à ça un tableau *_index* faisant le lien entre les index du solver et ceux du problème maître a été ajouté en attribut à la classe *GlpkSolver*, et les méthodes permettant de mettre à jour ce tableau ont été créées. Ce tableau, présent aussi dans les classes permettant de faire le lien avec Cplex, m'a de plus permis de trouver par la suite pourquoi ma première version de la classe *GlpkSolver* ne fonctionnait pas correctement.

En effet il existe un décalage sur les numéros des index des colonnes créées entre Cplex et Glpk, chose qui n'est mentionnée nulle part. Ainsi les index des colonnes commencent à 0 pour Cplex, alors qu'ils commencent à 1 pour Glpk. Ce petit détail, qui rendait ainsi le tableau *_index* obsolète, créait un décalage entre les colonnes réellement fournies au programme et celles qui devaient être

fournies en théorie. Le problème maître ne recevait ainsi pas la bonne solution, et le programme ne marchait donc pas correctement. Le problème a cependant été vite réglé une fois ce détail remarqué.

4.2.2 Avec stabilisation

La classe GlpkSolverStabilized a elle aussi posé quelques problèmes. En effet dans cette classe, qui sert à faire le lien avec le solver lorsqu'on utilise la stabilisation, le problème à résoudre, et donc à transcrire, contient beaucoup plus de contraintes que celui non stabilisé. L'initialisation du problème en est dont d'autant plus compliquée. Ainsi, en plus de créer les colonnes et les contraintes « de bases », à savoir pas plus d'une colonne par point et pas plus de p clusters au final, il faut aussi ajouter pour chaque point les bornes à ne pas dépasser, soit 2.n contraintes supplémentaires.

Ces contraintes supplémentaires se retrouvent aussi au moment d'envoyer la solution trouvée par le solver au programme. Il faut en effet récupérer les bonnes variables à renvoyer, sous peine de fausser totalement les résultats. Il a donc fallu faire très attention aux indices utilisés lors des manipulations sur les colonnes.

La classe *GlpkSolverStabilized* contient de plus des méthodes que la classe *GlpkSolver* n'a pas, telles les méthodes pour fixer les bornes des points, ou pour récupérer les diverses valeurs, telles les dites bornes.

4.3 Résultats du solver

Pour pouvoir comparer les deux solvers j'ai lancé un grand nombre de tests sur les critères de la clique, de l'étoile et de la coupe normalisée, critères gérés aussi par le programme. Je n'ai pu en faire sur la sommes des diamètres car, comme nous l'avons vu précédemment, la méthode exacte de ce critère est limitée à un faible nombre de points avec Glpk.

Les tests ont donc été effectués comme suit :

- sur un fichier de 59 points classés de 3 à 35 clusters pour le critère de la clique ;
- sur un fichier de 500 points classés de 50 à 250 clusters pour les critères de l'étoile et de la coupe normalisée ;

J'ai lancé en parallèle le programme avec Cplex et avec Glpk, avec et sans stabilisation, et est ensuite comparé différents résultats, qui ont été assez surprenants.

Ainsi pour le critère de la clique nous avons les résultats suivants : sans stabilisation le programme est de 10 à 50% plus lent en utilisant Glpk plutôt que Cplex. Cependant le nombre de colonnes générées par le programme et le nombre d'itérations qu'il lui a fallu pour trouver la bonne solution est

sensiblement identique, peu importe le solver utilisé. Une fois la stabilisation utilisée par contre, le programme devient aussi rapide que se soit avec Cplex ou avec Glpk.

Pour le critère de l'étoile Glpk est plus lent que Cplex, de 50% lorsque la stabilisation n'est pas utilisée, et de 10 à 20 secondes lorsqu'elle l'est. Cependant les temps totaux dépassent rarement la minute, donc cela reste acceptable.

Pour le critère de la coupe normalisée enfin, aucun solver ne se démarque. En effet le programme peut être un coup plus rapide avec Glpk, et le coup d'après 30% plus lent avec le même solver. Ces variations n'ont pu être expliquées.

On peut donc voir que dans l'ensemble Glpk obtient des résultats satisfaisants par rapport à ceux de Cplex, pour selon qu'il est gratuit. L'implémentation de l'utilisation de ce solver est donc un succès.

Cinquième partie

Méthode hiérarchique

5 Méthode hiérarchique

En parallèle du programme principal, nous avons décidé d'implémenter un petit programme indépendant permettant d'effectuer une méthode hiérarchique.

5.1 Principe et code existant

5.1.1 Principe

Le principe d'une méthode hiérarchique est le suivant : pour un ensemble de n points, on crée n clusters. On fusionne ensuite les clusters deux à deux de telle sorte que le coût de la fusion soit le plus faible possible, et ce jusqu'à n'obtenir plus qu'un seul cluster. Une telle méthode permet de savoir en combien de clusters il faut classer les n points pour avoir la meilleure solution optimale possible.

La fusion d'un cluster peut se faire de plusieurs façon différentes. Voici les deux principales méthodes existantes, méthodes que j'ai implémentées dans mon programme :

- La méthode du lien simple : lors de la fusion de deux points ou de deux clusters, on garde la plus petite dissimilarité entre cette nouvelle entité et les autres ;
- La méthode du lien complet : lors de la fusion on garde la plus grande dissimilarité entre la nouvelle entité et les autres ;

Pierre Hansen m'a appris qu'un programme permettant de faire une classification hiérarchique a déjà été crée par M. Fionn Murtagh, professeur à l'université de Londres, en Fortran 77. Il m'a donc conseillé de rechercher le code et de m'en inspirer pour créer mon programme, qui lui devait être en C++.

M. Hansen m'a également donner les références d'un article de M. Murtagh ([9]) dans lequel il explique les algorithmes qu'il a utilisé pour la création de son programme.

5.1.2 Code existant

L'algorithme principale utilisé par M. Murtagh est le suivant :

Étape 1 : construire la matrice de dissimilarités ;
Étape 2 : construire une chaîne de NNs à partir d'un point ; (NN : Nearest Neighbour)
Étape 3 : Fusionner les RNNs en un cluster ; (RNN : Related NN)
Étape 4 : mise à jour des dissimilarités et de la liste des NNs ;
Étape 5 : Recommencer l'étape 2 jusqu'à n'avoir qu'un seul cluster

Figure 25 : Algorithme de la méthode hiérarchique

La matrice de dissimilarité contient les dissimilarités entre les différents points, basées sur la norme euclidienne. Ainsi la dissimilarité entre deux points X(x1, .. , xn) et Y(y1, .. , yn) est calculée de la manière suivante :

$$d(X,Y)=\sum_{1}^{n}(x_i-y_i)^2$$

Pour l'étape 2, la chaîne de Nearest Neighbour, ou voisin proche, et construite selon l'algorithme suivant :

On prend un point i au hasard ;
On prend j = NN(i) ;
On continue jusqu'à trouver p = NN(q) et q = NN(p) ;

Figure 26 : Algorithme de construction de la liste des NN

Lorsque p = NN(q) et q = NN(p) on dit que les points p et q sont des RNN, Related Nearest Neigbhour, soit voisins proches reliés. On définie de plus la relation j = NN(i) comme suit :

$$j=NN(i) \;\rightarrow\; d(i,j)=min(d(i,k)),\quad k\in P\setminus\{i\}$$

où P est l'ensemble des points.

La programme de M. Murtagh affiche en sortie un dendogramme qui permet de connaître le coût des fusions.

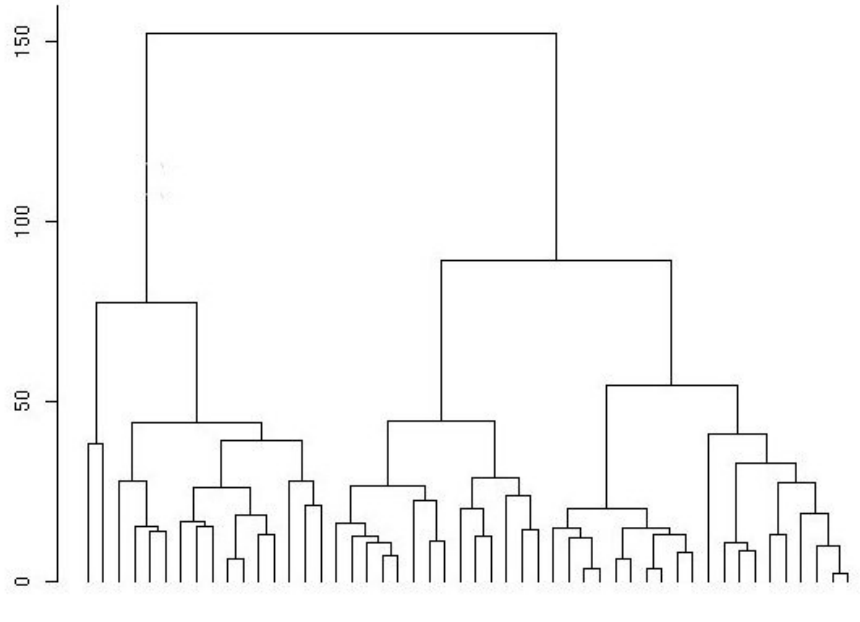

Figure 27 : Exemple de dendogramme

Sur ce dendogramme l'axe des abscisses représente les points ou clusters à fusionner, et l'axe des ordonnées représente le coût engendré par la fusion des ces deux entités. Avec un tel dendogramme on peut se rendre compte que pour cet exemple une classification en 4 clusters est bien alors qu'une classification en 1, en 2 voire en 3 clusters engendre des coût très élevés.

5.2 Implémentation

Je me suis donc inspiré du code de M. Murtagh pour créer le mien. J'ai ainsi créé plusieurs méthodes, ayant chacune un objectif bien précis :

- la méthode *readFile*, qui permet de lire le fichier de points passé en paramètre du programme et de stocker les points dans une matrice. Cette méthode a été reprise du programme principal ;
- la méthode *buildDissMatrix*, qui construit la matrice de dissimilarités à partir de la matrice de points ;
- la méthode *run*, qui est la méthode principale, et dont le fonctionnement sera détaillé par la suite ;
- la méthode *ConstructList*, qui permet de construire la liste des NNs ;
- la méthode DetermineNN, qui permet de choisir quelles entités fusionner ;
- la méthode *updateDiss*, qui permet de mettre la matrice de dissimilarités à jour ;
- la méthode *updateNN*, qui permet de mettre à jour la liste des NNs ;
- la méthode *display*, qui affiche le résultat ;

Le choix des structures de données utilisées s'est fait en fonction des avantages et des inconvénients de chaque type, et de l'utilisation que j'allais en faire. Ainsi la matrice de dissimilarités, qui était dans le code de M. Murtagh une matrice triangulaire stockée dans un simple vecteur, est dans mon programme une véritable matrice, symétrique, stockée dans un vecteur de vecteur. Ce choix a été fait pour faciliter l'accès aux données, évitant ainsi l'utilisation d'une méthode devant calculer la position du point dans la matrice grâce aux coordonnées de ce point, comme cela était fait dans le code en Fortran.

De même la liste de NNs est en fait un vecteur, là aussi par soucis d'accès au données. J'ai enfin utilisé un vecteur pour stocker les dissimilarités associées aux NNs, appelé *dissnn*.

La méthode run fonctionne selon l'algorithme suivant :

```
Cpt = nombre de points
on construit la liste des NNs

tant que cpt est différent de 1
    cpt = cpt – 1
    on choisi quelles entités fusionner
    on met à jour les données pour la méthode display
    on met à jour la matrice de dissimilarités
    on met à jour la liste des NNs
fin tant que
```

Figure 28 : Algorithme de la méthode run

La méthode qui a posé le plus de problèmes s'est avérée être la méthode de mise à jour de la matrice de dissimilarités. En effet il fallait mettre à jour les bonnes cases et avec les bonnes valeurs, le tout suivant le critère utilisé (lien simple ou lien double).

Car si au départ la fusion de deux points i et j ne pose pas de problème particulier, la fusion de deux clusters contenant chacun plusieurs points peut s'avérer quand à elle plus délicate. Il a donc fallu trouver comment faire en sorte que la fusion de deux clusters soit identique, au moins dans la forme, à la fusion de deux points.

Cela a été fait en utilisant un vecteur qui m'a permis de marquer les entités sélectionnées. Ainsi lors de la fusion de deux points, l'un des deux est marqué comme étant fusionné et ne sera plus utilisé. Il ne sert donc à rien de mettre à jour ses dissimilarités. Et grâce à cette méthode les clusters sont considérés comme des points par la méthode de mise à jour, et la fusion de deux clusters se fait facilement.

La sélection des NNs à fusionner se fait selon l'algorithme suivant, les points i_m et j_m étant ceux sélectionnés :

```
dmin = 100.000

pour chaque point i
    si le point n'a pas déjà été fusionné avec un autre
       si dissnn(i) < dmin
          dmin = dissnn(i)
          im = NN(i)
          jm = i
        fin si
     fin si
fin pour

retourner im et jm
```

Figure 29 : Algorithme de la méthode DetermineNN

5.3 Résultats

Comme on l'a vu le programme de M. Murtagh affiche un dendogramme. Cependant pour ne pas passer trop de temps à coder la méthode *display*, nous avons décidé de ne pas chercher à faire de même mais plutôt à afficher un tableau qui comprendrait les entités à fusionner, le coût de la fusion et le nombre de clusters. Voici un exemple de ce que nous fourni le programme :

```
Cost  NbCluster  Points
1     19   X X . . . . . . . . . . . . . . . . . . .
1     18   X . X . . . . . . . . . . . . . . . . . .
1     17   X . . . . . . . . . X . . . . . . . . . .
1     16   X . . . . . . . . . . . X . . . . . . . .
1     15   X . . . . . . . . . . X . . . . . . . . .
1     14   . . . X X . . . . . . . . . . . . . . . .
1     13   . . . X . X . . . . . . . . . . . . . . .
1     12   . . . X . . X . . . . . . . . . . . . . .
1     11   . . . X . . . X . . . . . . . . . . . . .
1     10   . . . X . . . . X . . . . . . . . . . . .
1     9    . . . X . . . . . X . . . . . . . . . . .
1     8    . . . X . . . . . . . . . X . . . . . . .
1     7    . . . X . . . . . . . . . . X . . . . . .
1     6    . . . X . . . . . . . . . . . X . . . . .
1     5    . . . X . . . . . . . . . . . . X . . . .
1     4    . . . X . . . . . . . . . . . . . X . . .
1     3    . . . X . . . . . . . . . . . . . . X . .
1     2    . . . X . . . . . . . . . . . . . . . X .
289   1    X . . X . . . . . . . . . . . . . . . . .
```

Figure 30 : Résultat de la méthode hiérarchique

La colonne cost représente le coût engendré par la fusion des deux entités sélectionnées, qui correspondent au 'X' dans le tableau. On remarque ainsi que pour ce fichier de 20 points on peut fusionner sans problèmes jusqu'à deux

clusters, mais la fusion en un unique cluster est très coûteuse.

Le programme fourni des résultats cohérents avec les tests que nous avions menés pour les différents critères, et ce pour tout nombre de points. De plus ces résultats sont très rapides. Il faut par exemple 13 secondes au programme pour traiter un fichier de 1.000 points.

L'implémentation d'une méthode hiérarchique a donc été un succès.

Conclusion

Le but du stage était de tenter d'optimiser les performances du programme existant, et de continuer son développement. Pour ma part j'ai plus précisément sur les parties concernant le critère de la somme des diamètres et du lien avec les logiciels de programmation linéaire. Il m'a de plus été demandé de coder une méthode hiérarchique indépendamment du projet.

Ces objectifs sont tous considérés comme atteints. L'optimisation des performances, qui s'est traduite par des modifications parfois mineures dans le code, a permis de faire gagner du temps à l'exécution du programme. La suite du développement du critère de la somme des diamètres, bien qu'apportant des résultats mitigés, a vu l'adaptation de la méthode exacte pour Glpk. L'ajout de la possibilité d'utiliser un logiciel de programmation linéaire gratuit ouvre des perspectives pour la mise à disposition du programme, et m'a permis de découvrir comment utiliser ces logiciels dans un programme. Enfin j'ai créé un programme permettant d'utiliser une méthode hiérarchique de manière totalement autonome.

Le fait de continuer un projet existant a présenté une certaine difficulté au début, car il nous a fallu bien assimiler tout le travail réalisé par nos prédécesseurs, et reprendre leur code. Néanmoins travailler sur un projet de cette envergure m'a permis de progresser au niveau de la vision globale d'un projet, ainsi que dans le langage C++ et en programmation linéaire. Enfin le stage m'a vraiment fait découvrir le travail en équipe.

Le programme n'est cependant pas fini, et sera continué par les prochains stagiaires. La création d'une heuristique initiale donnant de meilleurs résultats que l'actuelle sera ainsi à faire pour le critère de la somme des diamètres, ainsi que des modifications importantes dans la structure du programme, tel par exemple l'ajout de la possibilité de ne pas utiliser de méthode heuristique.

Nous savions dès le départ que le programme serait continué par d'autres stagiaires, et nous avons donc tout fait au cours de notre stage pour que cela se fasse le plus simplement possible, en commentant correctement notre code et en créant des tutoriels expliquant les parties les plus compliquées.

Références bibliographiques

[1] http://www.gerad.ca, site internet du GERAD

[2] Helfenstein R., Rapport de stage de 2e année de l'ISIMA, 2008

[3] Legrand B., Rapport de stage de 2e année de l'ISIMA, 2008

[4] Ruiz M., Rapport de stage de 2e année de l'ISIMA, 2008

[5] Sabatier M., Rapport de stage de 2e année de l'ISIMA, 2008

[6] Thouement S., Rapport de stage de 2e année de l'ISIMA, 2008

[7] http://www.gnu.org/software/glpk/, site internet du logiciel Glpk

[8] Fischetti M. & Baracco D., Interfacing a MIP heuristic based on ILOG Cplex with different LP solvers, 2004

[9] Murtagh F., Complexities of hierarchic clustering algorithms: state of the art, Computational Statistics Quarterly, 1, 101-113, 1984.

Lexique de programmation

Attribut :
selon l'orientation objet un attribut est une donnée interne à une classe.

C :
langage de programmation créé dans les années 1970.

C++ :
langage de programmation objet créé dans les années 1980 par Bjarne Stroustrup basé sur le C.

Classe :
description abstraite des données et du comportement d'un objet.

Diagramme de classe :
diagramme UML représentant les liens des classes les unes par rapport aux autres.

Fichier d'en-tête :
en C++, fichier contenant une liste des libraires utilisées et la définition de la classe concernée.

Fortran 77 :
langage de programmation créé en 1956 utilisé principalement pour les mathématiques et les applications de calculs scientifiques.

Générateur de nombre pseudo aléatoire :
algorithme générant une séquence de nombres présentant des propriétés du hasard.

Interface :
classe abstraite, c'est à dire qui ne peut être instanciée, qui définie les grandes lignes à adopter par les classes en dérivant.

Librairie :
ou bibliothèque : ensemble de fonctions ou de procédures, ayant de préférence un point commun, mises à la disposition des programmeurs.

Logiciel libre :
logiciel dont la licence est libre, que chacun peut utiliser, modifier, étudier et diffuser.

Logiciel propriétaire :
logiciel entravant l'une des possibilités d'un logiciel libre ; le terme logiciel propriétaire est souvent employé pour désigner un logiciel dont l'utilisation requiert l'achat d'une licence auprès de son producteur.

Main :
programme principal.

Makefile :
logiciel traditionnel d'UNIX servant à créer des fichiers.

Méthode :
selon l'orientation objet, une méthode est une fonction interne aux classes.

Pointeur :
en C++, comme en C, un pointeur est une variable contenant l'adresse mémoire de l'objet « pointé ».

UML :
Unified Modeling Language, ou Langage de Modélisation Unifié ; c'est un langage graphique non propriétaire de modélisation des données.

Lexique du programme

Cluster :
terme anglais pour classe ; ensemble de points. On parle aussi de colonne.

Dissimilarité :
représente les différences entre deux points ; lorsqu'elle est basée sur la norme euclidienne il s'agit de la distance entre les deux points.

Génération de colonne :
méthode permettant d'initialiser les programmes linéaires efficacement.

Méthode exacte :
méthode permettant de trouver à coup sur, s'il en existe, une colonne améliorante pour le problème. Dépend de chaque critère.

Méthode heuristique :
méthode approximative permettant de trouver des colonnes améliorantes pour le problème. Est plus rapide que la méthode exacte.

Partition :
une partition de n points en p clusters signifie que chaque point est dans un et un seul cluster.

Problème maître :
partie commune à tous les critères, elle s'occupe de faire le lien entre autre entre le sous problème, les classes de stockage des données et les solvers.

Solver :
logiciel de résolution de problèmes linéaire, tels Cplex ou Glpk.

Sous problème :
partie propre à chaque critère, elle s'occupe en autre de calculer le coût d'une colonne, ou de résoudre le problème de manière exacte.

Stabilisation :
méthode permettant de cibler les solutions recherchées par le programme.

Stochastique :
se dit de quelque chose qui n'est pas déterministe, c'est à dire dont on ne peut prédire le résultat final, qui dépend du hasard.